Implementation and Performance Evaluation of the Topic-Specific Trust Open Rating System

Holger Lewen

Institute AIFB, Universität Karlsruhe (TH), Germany
lewen@aifb.uni-karlsruhe.de

Abstract. In this technical report we describe the implementation of the Topic-Specific Open Rating System (TS-ORS) based on the use-case of integrating it with the Open University's Semantic Web gateway Watson within the Cupboard system. Design decision are explained and the performance is evaluated on two different machines. Then the results are analyzed and recommendations given for people employing the system.

1 Introduction

After providing the theoretical details of the topic-specific open rating system (TS-ORS) in [1], this technical report deals with a concrete implementation of the system[1]. The concrete implementation described hereafter is part of the integration of the TS-ORS and the Open University's Semantic Web gateway Watson within the Cupboard system[2], and thus was implemented with concrete application requirements in mind.

The TS-ORS allows ranking of ontologies based on user reviews and trust users express in these reviews. Everybody can share their opinion about an ontology–property combination and these reviews will be displayed to other users. A review consists of a 5-star rating and a justification for that rating. Other users can then state whether they find the review helpful or not and thus express whether they trust or distrust the reviewers. Based on the trust information, a web of trust is computed that allows the ranking of reviews for each ontology–property combination. It allows a personalized ranking for users that are logged in, and a ranking based on common opinion for users not identifiable or not connected to the web of trust. Based on the top review(s) for each ontology–property combination, a linear combination of the star ratings of the best reviews for all different properties allows to compute an overall rating for the ontology. Using these scores, the ontologies can be ranked.

The Watson system[2] can be seen as a search engine for the Semantic Web. It supports users and applications in finding, selecting and (re)using ontologies that are available online, through advanced search and exploration mechanisms.

Combining the TS-ORS and Watson allows users of Watson to directly retrieve quality information on ontologies stored in Watson (in case somebody

[1] SourceCode is available on request by contacting lewen'@'aifb.uni-karlsruhe.de
[2] http://watson.kmi.open.ac.uk

reviewed that ontology). A common problem when reusing ontologies or parts of ontologies, is that the quality is unknown. Because the number of ontologies one could possibly reuse can be large, it is impossible to assess all ontologies by yourself. For that reason help from the community is needed. Furthermore, the integration allows Watson users to affect the ranking of results by changing parameters and expressing trust. The code developed for that scenario was then also used for the Cupboard system, which introduces the notion of ontology spaces for each user, where ontologies can be uploaded and also reviewed by the community.

Based on the scenario described above some concrete design decisions were made, which will be described in the following section. The section is structured as follows: First design decisions are explained, then the overall architecture and concrete implementation details are presented. Lastly the results of a performance evaluation are discussed.

2 Implementation

2.1 Design Decisions

The functionality needed for Watson was basically storing and retrieving reviews and providing ranking and rating results. Furthermore, meta trust statements are supported, but only on the level of ontologies, users and properties, since Watson does not support domain information yet. This feature can however easily be added at a later point, since it only affects the meta trust propagation. So in terms of trust statements mentioned in [1], only those displayed in table 1 are allowed in this concrete implementation. In terms of interaction, the TS-ORS takes care of assigning the internal IDs needed for computation and storage (see section 2.2. It takes as input URIs as identifiers for ontologies and is flexible in terms of user identification. It has its own MySQL database to store reviews, users, ontologies and trust information. The application is implemented in JAVA, and servlets are used for interacting with Watson and other programs. At the moment, REST services are offered that provide the results either as JSON, XML or HTML, based on content negotiation using the HTTP header.

Table 1. Allowed Metatrust Statements

Statement	**Scope**	**Explanation**
W	$A_i \times A_j \times O_n \times X_k \times D^{O_n} \rightarrow T_u$	Statement on a specific property of a specific ontology
$W_{O_n X}$	$A_i \times A_j \times O_n \rightarrow T_u$	Statement on all properties of a specific ontology
W_{CX_k}	$A_i \times A_j \times X_k \rightarrow T_u$	Statement on a specific property of all ontologies
W_{CX}	$A_i \times A_j \rightarrow T_u$	Statement on all properties of all ontologies

2.2 Architecture

The first design decision concerned the optimal design of the database schema with regard to retrieval and partitioning of information.

Database Schema One of the foremost concerns when using databases in an application is developing a database schema that ensures data integrity but does not sacrifice performance. In order to optimize performance, it is necessary to use database indexes and ensure sufficient memory is allocated for caching. In our case we decided to store the most basic information, like users, ontology properties, ontologies, ratings, trust between users, and meta trust in dedicated tables (see figure. 1). We chose MySQL as a database with MyISAM storage engine for better performance. We will now describe the different tables:

- **Users**: This table stores the relation between the user ID (uid) and user data.
- **Ontologies**: This table stores the relation between the ontology ID (oid) and the ontology uri.
- **Properties**: This table stores the relation between the property ID (xid) and the name of that property.
- **Rating**: This table stores the relation between the rating ID (rid) and the review. For each review, a combination of uid, oid and xid exists along with the review data $dext$ (textual part of the review) and $dstar$ (the 5 star value).
- **Trust**: stores the relation between the trust ID (tid) and the trust statement. For each trust statement, a combination of uid, rid exists along with the information whether trust or distrust was expressed.
- **Metatrust**: stores the relation between the meta trust ID ($mtid$) and the meta trust statement. Three different meta trust statements are possible (also see table 1). Depending on the type of meta trust statement, $global, ontology$ or $property$ is set to 1 in case of meta trust and -1 in case of meta distrust.
- **Runtimetemp**: During the initial computation of $globaltrust$ and $localtrust$, a table called temp is temporarily created. This table is renamed "runtime-temp" after the computations are completed. The only reason for having the table is optimizing performance. The table is basically the result of joining the trust and rating table. This materialization saves time because specific indexes can be created and the join only has to be performed once. As long as there are not two separate databases for runtime data and offline data, the database will be in use when the recomputation of trust is triggered. So during creation, the fresh data has to be stored in temporary tables that can later replace the current runtime tables.
- **Globaltrust**: This table stores the trustrank and distrustrank of users for a given oid, xid combination. To improve runtime-performance, also the dstar and rid of the review are stored in this table. During the initial computation, a table tempglobal trust with identical schema is used, which is renamed global trust once the computations are complete. The need for a runtime and a temporary table is also due to the fact that otherwise small updates could not be directly displayed without recomputing everything.

– **local trust**: This tables stores the local trust value along with its interpretation as trust or distrust for a given oid, xid combination. To improve runtimeperformance, dstar and rid of the reviews are also stored in the table. During the initial computation, a table templocal trust with identical schema is used, which is renamed local trust after the computations are complete.

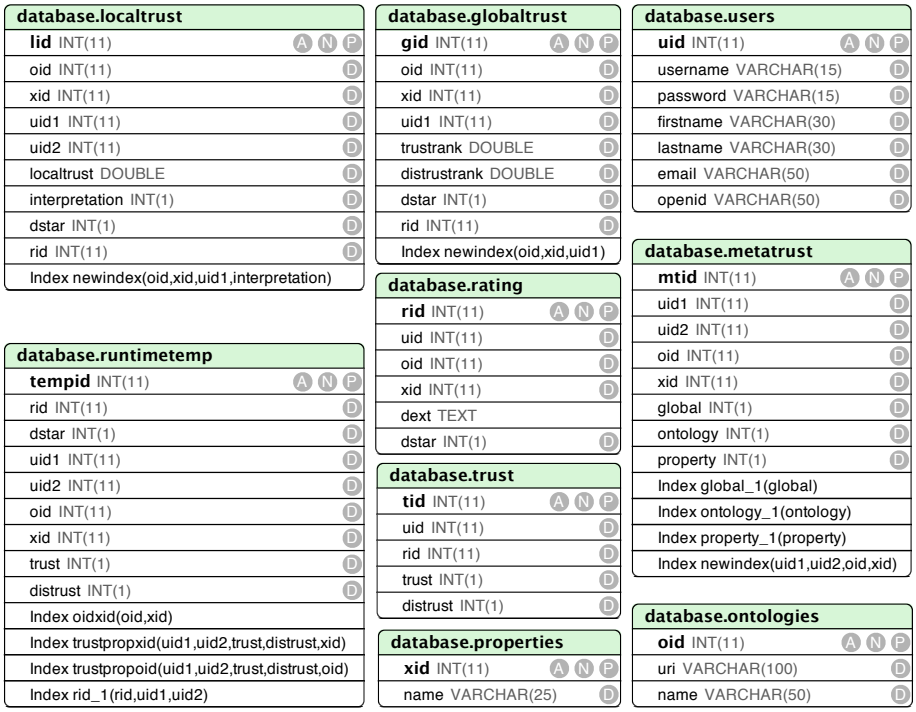

Fig. 1. This graphic depicts the database schema used for the implementation of the TS-ORS.

2.3 UML-Diagram

The methods and functionality were partitioned into different classes. We will first describe the classes and methods of the core package and then the classes needed for the interaction with external applications.

TS-ORS Core As can be seen in figures 2 and 3, the functionality is distributed among different classes. They are grouped roughly by their functionality. The classes used for the most basic databank interaction are not displayed, since they only provide basic functionality like asking SQL queries and performing SQL

updates or getting a connection from the connection pool. We will now quickly describe the different classes. If a method contains the string "update" it means that it performs the changes as well on the runtime databases, so the result of the operation is usually visible immediately instead of only after recomputation of trust values.

- **DBInteraction**: This class contains methods that primarily read and write information from the database. Most methods simply encapsulate SQL queries or update statements. Some variables are used to strore results in between method calls. The behavior of most methods can be inferred from their signature. It is used by methods from other classes for retrieving the needed information from the database and writing the results back.
- **Computations**: This class contains methods that perform the computations needed for determining trustrank, distrustrank, local Trust as well as ranking order and overall rating computation. It also contains methods for performing the majority rounding that has to be performed as part of the local trust computation.
- **Multithreading**: In order to distribute the computation of local trust and global trust (trustrank and distrustrank) among different threads, this class contains all the methods needed for performing the computations and a *run* method to invoke the thread.
- **Metatrust**: This class contains methods that are needed to set and retrieve meta trust statements (see table 1). The method *setEigentrust* is used to ensure that every reviewer trusts his reviews. The *propagateMetaTrustand-Distrust* method distributes the meta trust statements down to the level of normal trust statements.
- **Setup**: This class contains methods that can be used to create and drop database tables and indexes.
- **Settings**: This class contains variables that are read by different methods. It is the main place to change settings of the application.
- **Updates**: This class contains methods that are used to alter and refresh data in the system. The method *recomputeEverything* is the main method that refreshes runtime-data based on the information of the database tables. Since not all changes can be directly computed at runtime, it is important to call this method at fixed timepoints (when exactly can be based on size of the system and frequency of changes).
- **GenerateBulkData**: This class contains methods used to generate random data that can be used to test the functionality of the system. Based on the parameters entered, the respective number of instances are created in the database. It is ensured that no duplicates are created.
- **CachedObjects**: This class contains objects that cache information from the database, so that database interaction can be minimized.

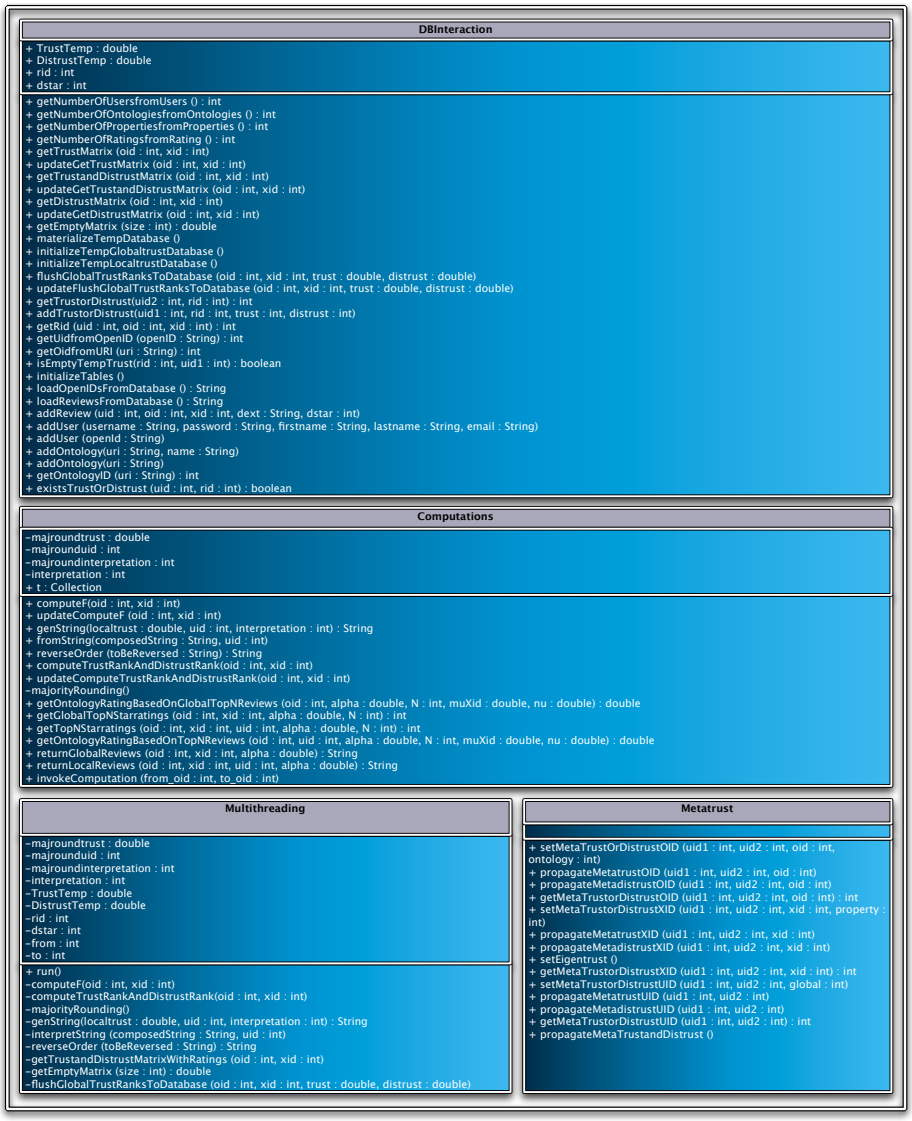

Fig. 2. This graphic depicts a UML class diagram of some of the TS-ORS core components.

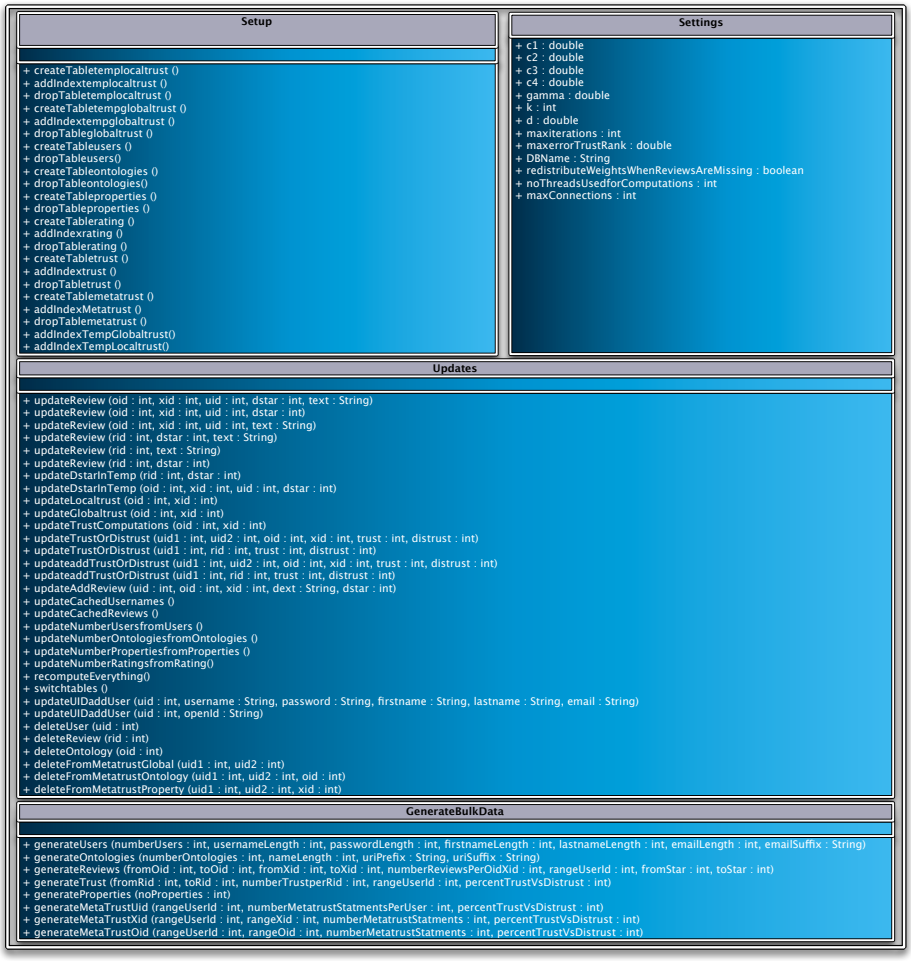

Fig. 3. This graphic depicts a UML class diagram of the rest of the TS-ORS core components.

2.4 Interaction with other Programs

As can be seen in figure 4, the interaction with Watson is managed through Java Servlets running on a Tomcat Server. There are 6 servlets exposing the functionality:

- **AddReview**: This servlet is called to add reviews to the database of the TS-ORS. In case the ontology is not known to the system, it is automatically added to the database.
- **AddTrustOrDistrust**: This servlet can add trust or distrust between two users for a specific ontology–property combination.
- **ManageMetatrust**: Using this servlet, meta trust information can be added to the database.
- **ReturnReviews**: Using this servlet, reviews can be retrieved for a certain ontology–property combination. If the user is logged in, they are returned in a personalized order, if not, they are returned based on global trust. Depending on content negotiation, the reviews are returned either as XML, JSON or HTML.
- **ReturnOverallRating**: Using this servlet, overall ratings are retrieved for the specified ontology and based on the parameters provided. If the user is logged in, they are returned in a personalized order, if not, they are returned based on global trust. Depending on content negotiation, the reviews are returned either as XML, JSON or HTML.
- **Admin**: This servlet can be used to access administrational functionality, such as triggering recomputation.

Furthermore, two classes called **Overallrating** and **Review** are used for storing the results before they are serialized as either XML or JSON.

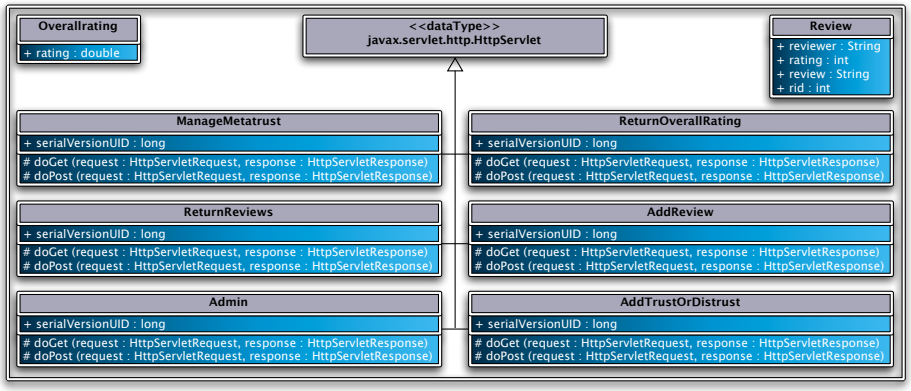

Fig. 4. This graphic depicts a UML class diagram of the servlets used for interaction with Watson.

3 Evaluation

Because the performance of the system is one of the most important aspects when serving a web application that provides results to the user, we have benchmarked the time to perform certain functionality. In order to get good estimates, we have seeded the tables with randomly generated data and could thus see how the number of users or trust statements or reviews affect the computation time.

3.1 Setup

We created a Java class called Evaluation that contains a method to run our evaluation setup. This way, people who have the code can run it and compare the results to the performance of their system. The method works by first filling the database with randomly generated data, based on predefined parameters. Parameters include the number of users, ontologies, reviews per ontology–property combination, trust statements on these reviews and meta trust statement as well as the percentage of trust vs. distrust. It is checked that no duplicate/contradictory data is generated during the process.

After the data is created, the computationally heavy steps are performed to compute the trustrank, distrustrank and local trust for all ontology–property combinations. This step also has to be performed from time to time in running implementations to make sure the most current data is used for the ranking of reviews. The process involves multiple steps:

- Materialization of the temp database to allow faster access
- Setting eigentrust (making sure every reviewer trusts his/her own review)
- Propagating meta trust and meta distrust statements (whilst ensuring that more specific statements are not overwritten by general statements)
- Computation of the trustrank, distrustrank and local trust (including interpretation) for all ontology–property combinations in the database (since this task also relies mainly on CPU and not only on database, it is parallelized)
- Indexes are added to tempglobal trust and templocal trust
- The tables are renamed: temp becomes runtimetemp, tempglobal trust becomes global trust, templocal trust becomes local trust. This way, the system can stay in operation when the trust information is recomputed.

Once this data is computed, the runtime tests are performed. We measure how long it takes to retrieve an overall rating for all ontologies, both using global and local trust. The tasks are performed 300 times based on a varying number of reviews (using only top, top 3 and all reviews). We measure the duration of each execution and report the minimum, maximum and average duration. It is important to see how database caches can be exploited to minimize the latency at runtime. The maximum time usually occurs when the rating is first retrieved and no result are cached, while for the rest of the runs the results of the database queries should still be cached. We can exploit the query result cache for short response time during runtime by querying all ontologies in the system once after

recomputation, so that most results are cached and runtime database access can be minimized. We also measure how long it takes to retrieve all reviews for all ontology–property combinations based on both global trust ranking and local trust ranking.

3.2 Results

We have performed the benchmark on two systems: A Dualcore MacBook Pro running Mac OS X 10.5.6 and a QuadCore PC running Vista 64bit.

In order to find out how a different number of reviews for each item and trust statements on these reviews would influence the computation time, we did not only vary the number of users (100, 250, 500), but also the amount of reviews and trust statements available, resulting in 9 different testruns. During the generation of the test data, for each different user group size, we had one setting which had 10& of the users review each ontology–property combination and 10% of the users then trusting or distrusting each of these reviews, one with a 50&–50& distribution and a worst case scenario (100%–100%). Worst case means that every users reviews every ontology–property combination, and also every user then votes on the usefulness of these reviews. In a realistic setting, the distribution is likely less than 10%–10%. We have furthermore assumed that of the users making trust-statements, 70% of the trust statements were trust, and 30% were distrust. For the metadatatrust generation, we have fixed the percentage of global, ontology- and property-specific meta trust statements to 20% per user for all runs, i.e. each users meta trusts 20% of the other users. The test data was regenerated between all runs, to prevent caching effects in between runs. As for the number of properties, they were fixed to 5. Furthermore we limited the number of ontologies to 12, since this was the smallest number allowing to distribute the computations evenly among 4 threads. Furthermore, the complexity of the computation does not increase by having more ontologies, this is just a linear factor (it will take 10 times longer to perform the computations for 120 instead of 12 ontologies). This is because all the computations have to be performed for each ontology–property combinations.

MacBook Pro The computation was performed on a MacBook Pro with 2.16 GHz Intel Core 2 Duo Processor and 3GB 667 MHz DDR2 SDRAM running Mac OS X 10.5.6. Harddisk is ST9320421ASG (320GB 7200 RPM, 16MB Cache, avg seek time 11 ms). Eclipse Version is 3.3.2 for Mac. MySQL versions are 5.0.45 for the server and 5.1.5 for the J-Connector.

The results are shown in the figures 5, 6, 7, 8 and 9. The percentages in the labels refer to the percentage of reviews, trust- and meta trust statements generated as test data (see explanation above). For example 10%10%20% means that 10% of all users have reviewed each ontology, 10% of all users have rated each review, and each user expresses meta trust towards 20% of the other users. We believe that this setup allows to draw some conclusions about the scalability of the system with regard to number of users and sparsity of data. In the following section we will analyze the results seen in the figures presented here.

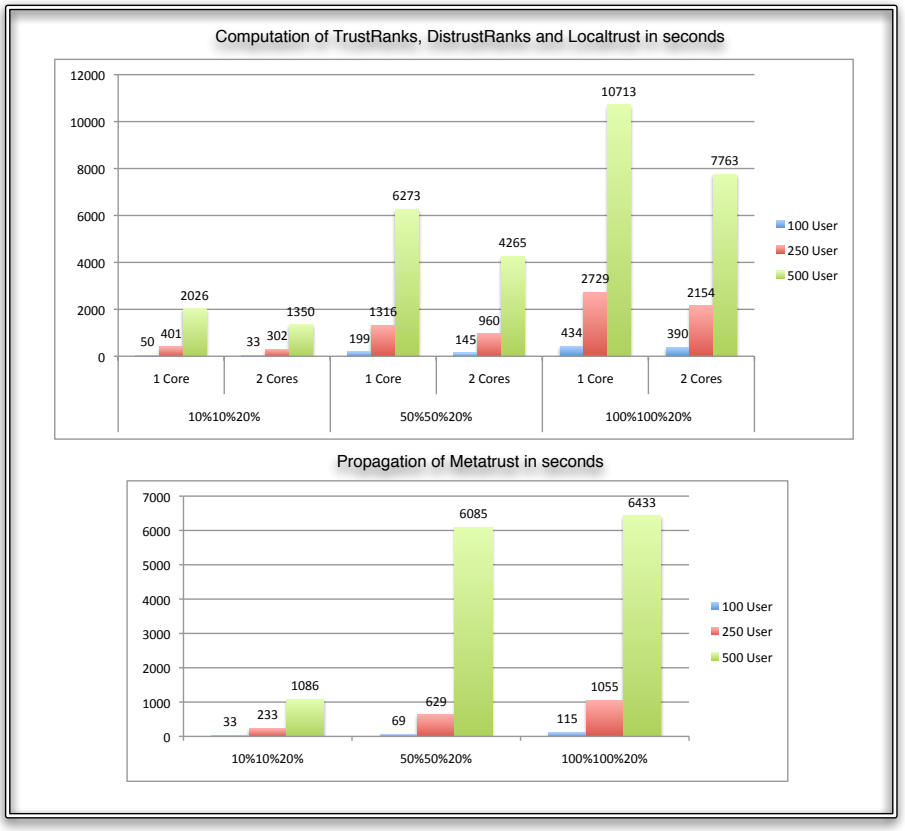

Fig. 5. This graphic depicts the results of the benchmark of the computation and meta trust propagation. Data is presented as time in seconds. (Run on MacBook Pro)

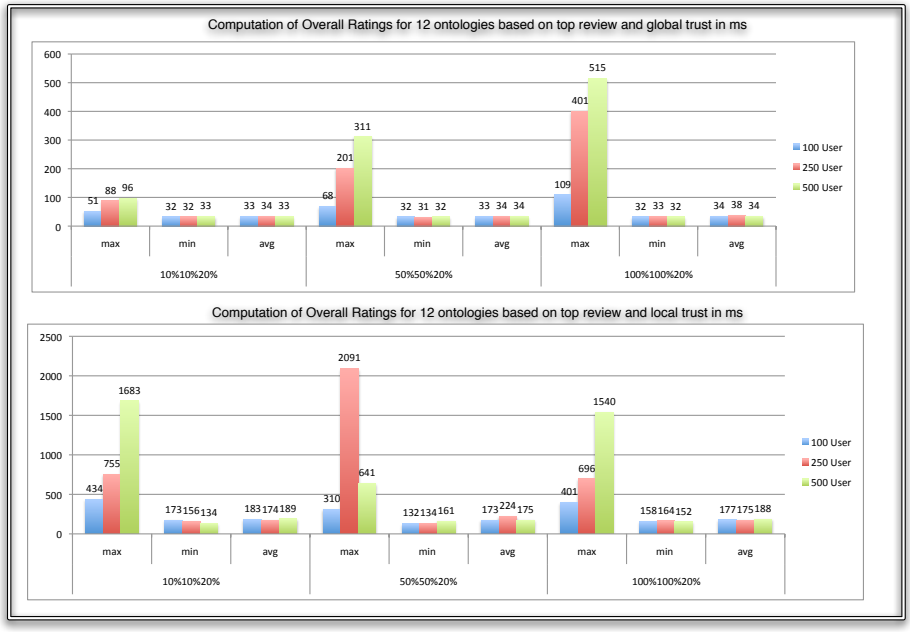

Fig. 6. This graphic presents the times taken (min, max and avg) for returning the overall rating for all 12 ontologies in ms once for global and once for local trust. 300 runs were performed and only the top review was considered for the computation. (Run on MacBook Pro)

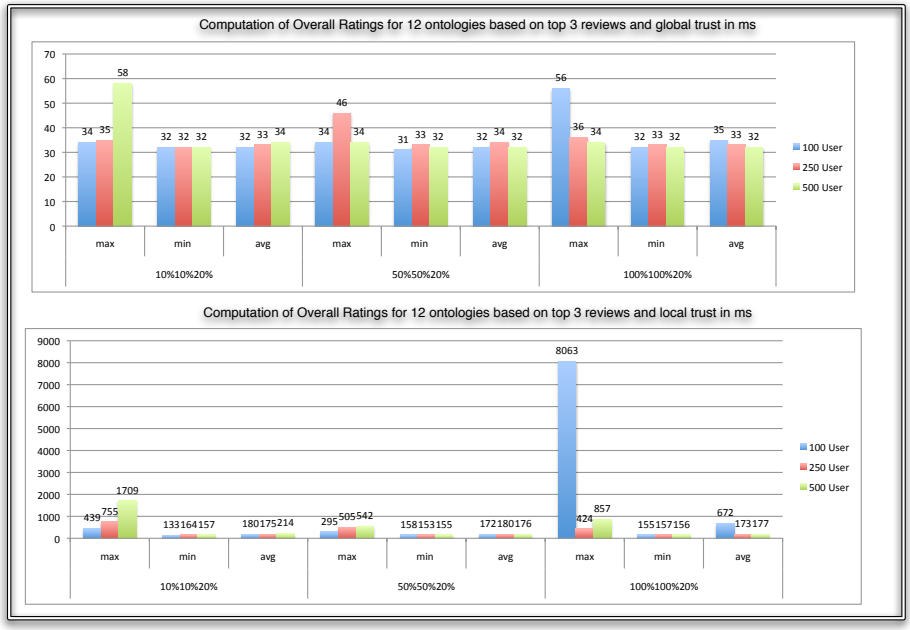

Fig. 7. This graphic presents the times taken (min, max and avg) for returning the overall rating for all 12 ontologies in ms once for global and once for local trust. 300 runs were performed and only the top 3 reviews was considered for the computation. (Run on MacBook Pro)

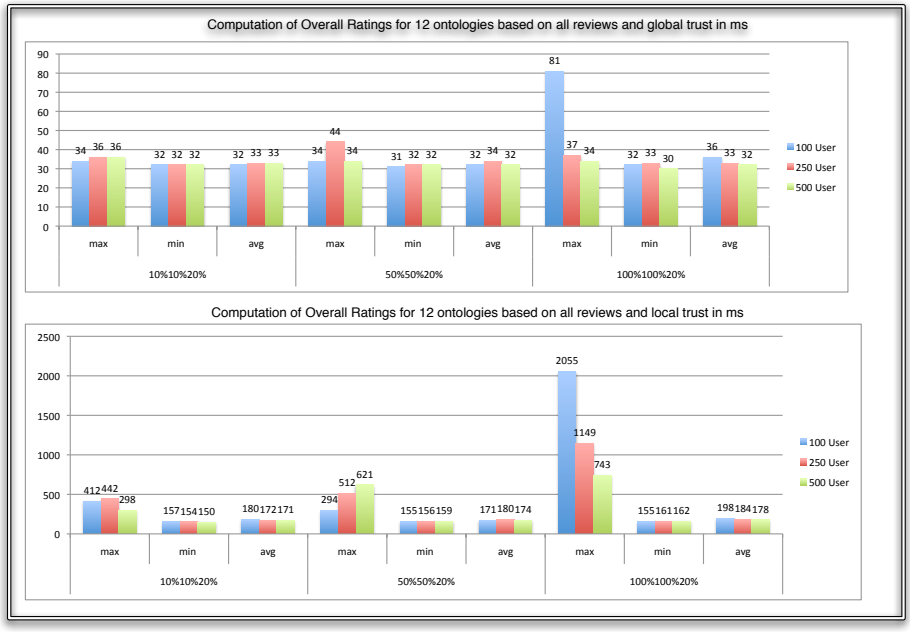

Fig. 8. This graphic presents the times taken (min, max and avg) for returning the overall rating for all 12 ontologies in ms once for global and once for local trust. 300 runs were performed and all reviews were considered for the computation. (Run on MacBook Pro)

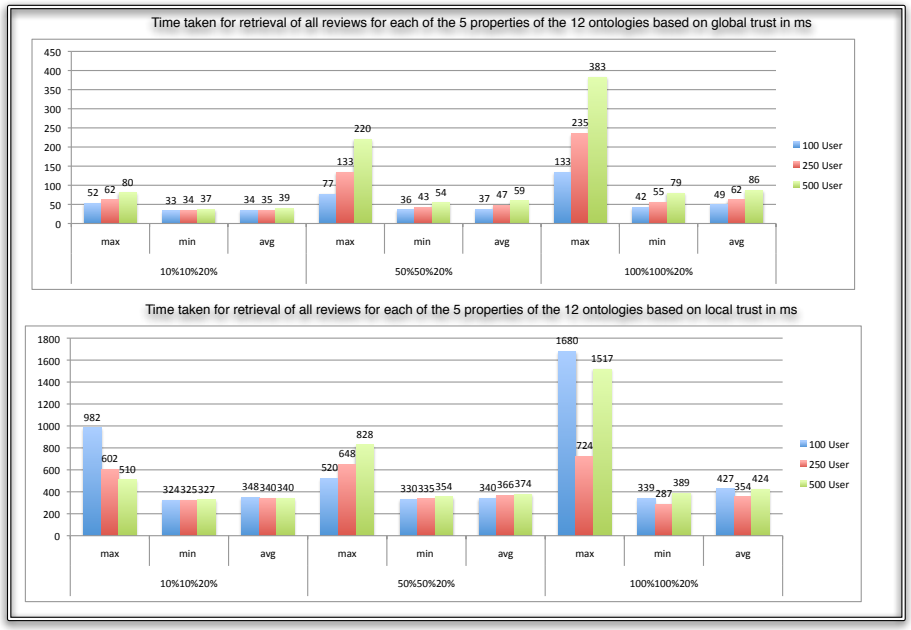

Fig. 9. This graphic presents the times taken (min, max and avg) for returning all reviews for all 5 properties of all 12 ontologies in ms once based on global and once based on local trust. 300 runs were performed. (Run on MacBook Pro)

QuadCore Vista 64bit The computation was performed on a 2.40 GHz Intel Core 2 Quad Q6600 processor with 8GB 800 MHz DDR2 SDRAM running Microsoft Vista Business 64bit. The harddisk is a Samsung HD103UJ (1TB, 7200 RPM, 32MB Cache, avg seek time 8.9). Eclipse Version is 3.4.1 for Windows. MySQL versions are 5.1.31 for the server and 5.1.7 for the J-Connector.

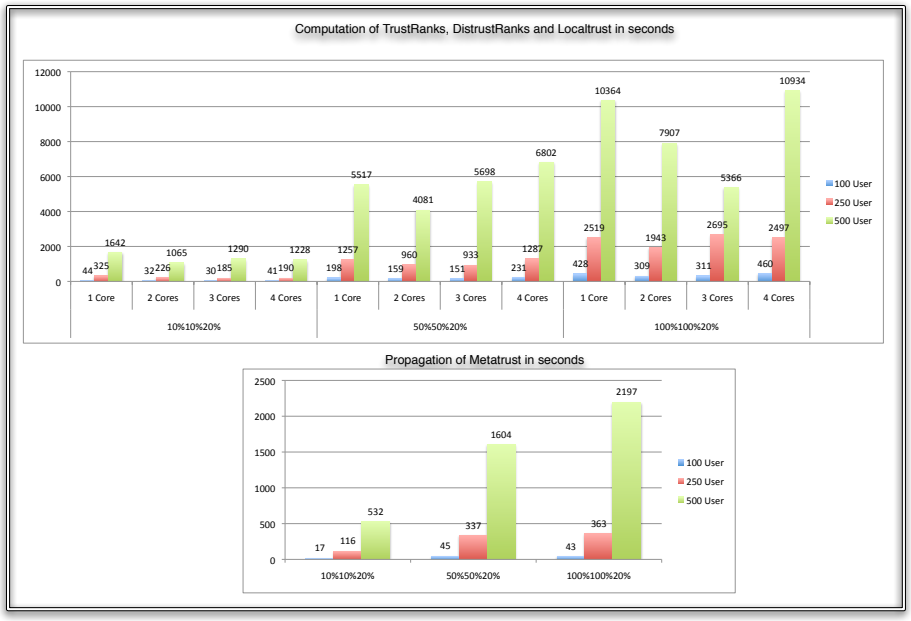

Fig. 10. This graphic depicts the results of the benchmark of the computation and meta trust propagation. Data is presented as time in seconds. (Run on Intel Core 2 Quad)

4 Result Analysis

We will now analyze the results of the benchmark and try to explain the system behavior. We will start with figures 5 and 10, which depict the computation of trust and the propagation of meta trust. In order to understand which steps are performed int these two events, we will now explain them in more detail.

4.1 Metatrust Propagation

The general idea behind the concept of meta trust is saving the user the time and effort to find all reviews by a particular user and state that he or she trusts them. So each meta trust statement can be seen as a number of particular trust

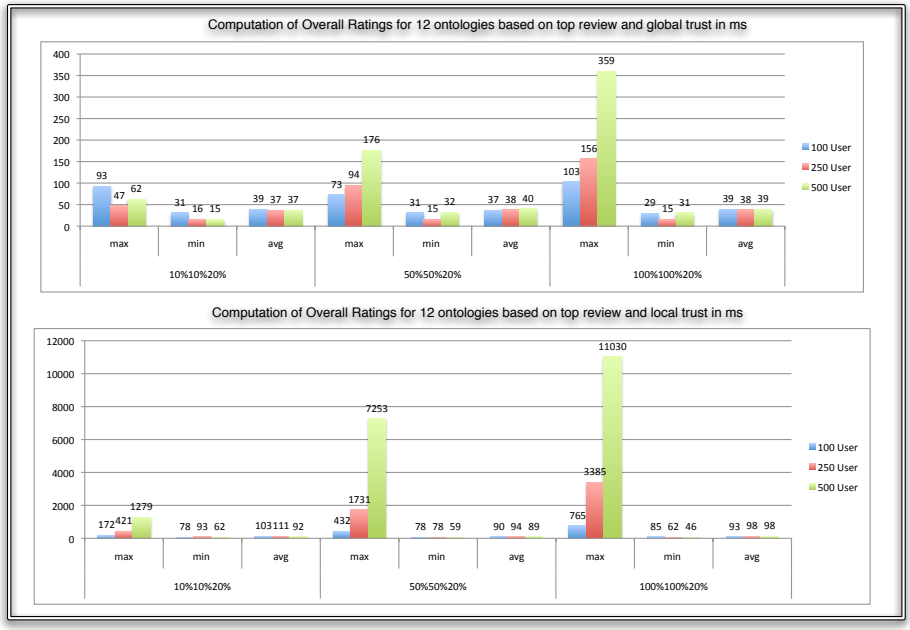

Fig. 11. This graphic presents the times taken (min, max and avg) for returning the overall rating for all 12 ontologies in ms once for global and once for local trust. 300 runs were performed and only the top review was considered for the computation. (Run on Intel Core 2 Quad)

Fig. 12. This graphic presents the times taken (min, max and avg) for returning the overall rating for all 12 ontologies in ms once for global and once for local trust. 300 runs were performed and only the top 3 reviews was considered for the computation. (Run on Intel Core 2 Quad)

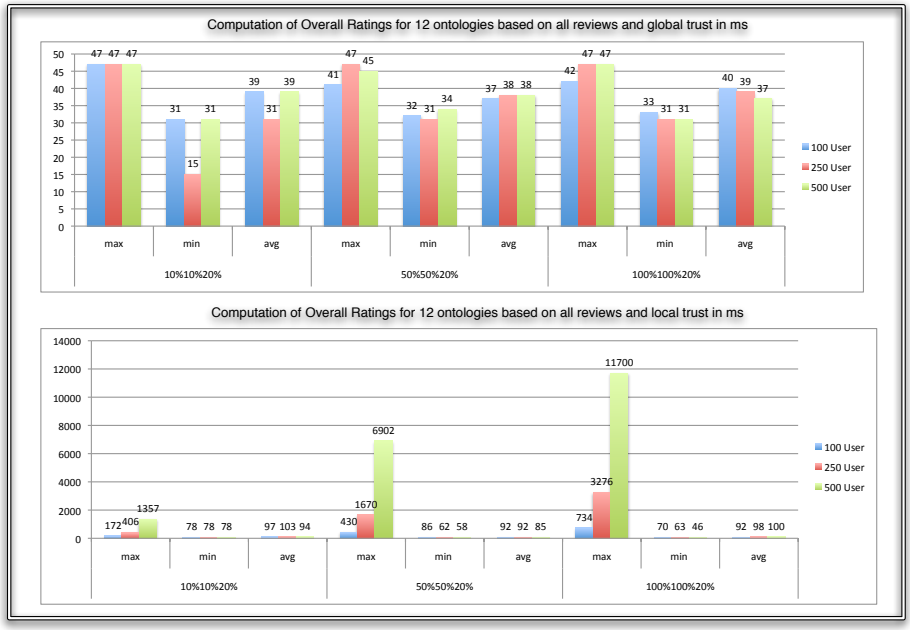

Fig. 13. This graphic presents the times taken (min, max and avg) for returning the overall rating for all 12 ontologies in ms once for global and once for local trust. 300 runs were performed and all reviews were considered for the computation. (Run on Intel Core 2 Quad)

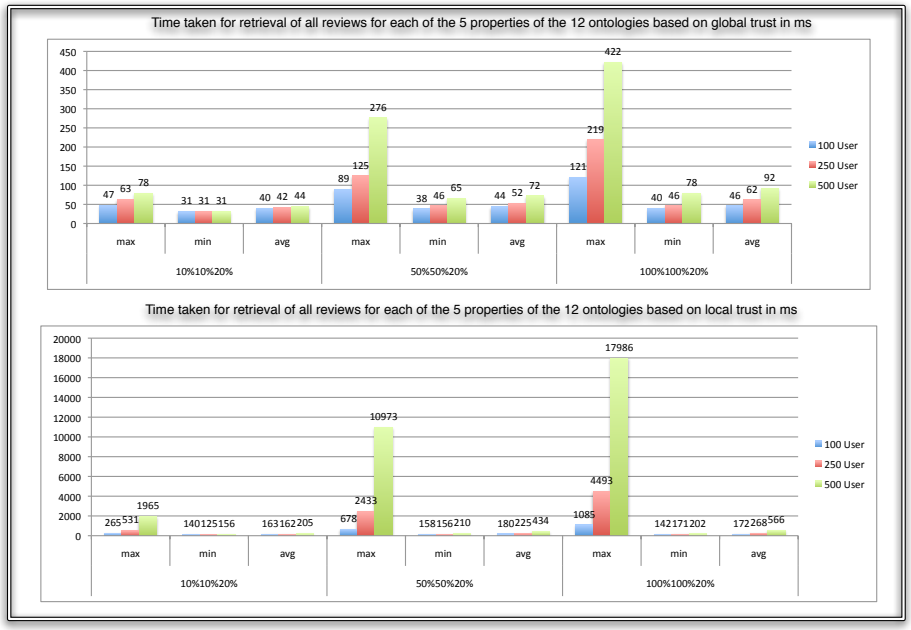

Fig. 14. This graphic presents the times taken (min, max and avg) for returning all reviews for all 5 properties of all 12 ontologies in ms once based on global and once based on local trust. 300 runs were performed. (Run on Intel Core 2 Quad)

statements towards the reviews covered by the meta trust statement. So if a meta trust statement is made for a particular user and ontology combination, the meta trust statement has to be transformed into several trust statements on all properties of the ontology this user reviewed. Another important conceptual decision is that more special trust or meta trust statements cannot be overwritten, and that distrust is more important than trust. This is important to allow constructs like trusting another user globally except for one ontology or property. In order to take all these limitations into account, the propagation of meta trust is performed as follows: First all meta distrust statements covering properties of ontologies are retrieved from the meta trust table and propagated. This is done by retrieving a list of reviews that the user has written for this property, and removing the reviews that are already trusted or distrusted by the user issuing the trust statement. Then for all the remaining reviews, a distrust statement is added. Afterwards, the same is done for meta trust statements covering the properties, meta distrust and -trust statements covering ontologies, and last the global meta distrust and -trust statements. As expected, the time taken to perform this operation increases with the number of users and the amount of trust statements already in the database. The main limiting factor here are the database lookups needed to check each time whether a more specific trust statement already exists, before propagating the meta trust.

The results indicate what was expected, namely that the duration increases if more reviews are in the system, and also if the number of users increases. Since this method almost solely relies on database operations, it can profit from the faster disk and faster disk access of the desktop harddisk. The computations are at least twice as fast on the better machine. In our example, the duration of execution is growing a little bit more than squared in the relation to the number of users. However, this is mainly due to the bigger number of reviews and trust statements that we set to a fixed portion of the users. So for the 10%10%20% that means that in the case of 500 users, we have 5 times the amount of reviews, trust- and meta trust statements. If we only increase the number of users, but not the number of reviews, the computation only lasts slightly longer. For example, the propagation of meta trust for the setting 500 users with a 2%, 2%, 4% setting (which is equivalent to the 100 User 10%10%20% setting) takes 144 seconds instead of 33 for the case of 100 Users (which is less than linear growth with regard to the increase in number of users). So in case the number of reviews is small in comparison to the userbase, an increase in users does not affect the computational time too much. The results for this comparison can be seen in figure 15.

4.2 Trust Computation

For the technical details of the computation, please see [1]. The benchmark results can be found in figure 5 and 10. The computations are the backbone of the whole TS-ORS system. After the meta trust is propagated, all trust statements are the most special form, i.e. on reviews of a specific ontology–property combination. In the trust computation phase, for each of the ontology–property

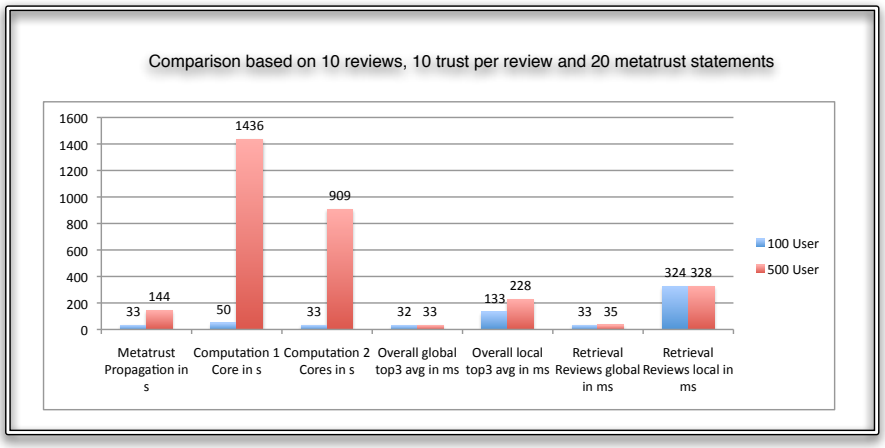

Fig. 15. This graphic presents a comparison between the time taken for different actions based on the same review and trust statements but increased number of users. (Run on MacBook Pro)

combinations, the trustrank, distrustrank and local trust are computed. Computing the trustrank is basically computing PageRank [3]. In order not to spend unnecessary time during the iterations needed to converge to a solution, we have implemented 2 stop criteria, which can be specified in the settings class. Either the overall change in trustrank (cumulated difference of ranks between two computations) is below a certain threshold, or a maximum number of iteration is reached.

The distrustrank can be computed in one iteration after the trustranks are known.

The local trust computation relies heavily on matrix multiplication. Also the results have to be interpreted, which is based on known trust and distrust statements and a comparison of unknown values to neighbors.

To see how multiple cores could share the computational effort, we have distributed the computation among up to 4 cores, and checked how time was affected. While the threads each had their own data in memory, they all accessed the same database (using a connection pool) to retrieve the trust and distrust information needed for the computation.

The results indicate that using 2 threads does indeed lead to faster computation. However, using 3 or 4 threads even on the 4 core machine mainly takes longer than just 2. It seems that the database access is the limiting factor here. If 4 threads try to read and write information from and to the database, they have to wait until the database can handle the request. So in this case having distributed databases that are able to handle the database interaction in a timely fashion would increase the performance more than having more threads on more CPUs.

The results seen in the figures are for computing all 60 ontology–property combinations. So you have to divide the results by 60 to know how long one trust computation cycle lasts. For updates during runtime (a single trust statement was added), it is possible to recompute the trust on the fly, since the time it takes for a realistic setting (few reviews and trust statements) should be less than 10 seconds (even under a second for the 100 user case with sparse data).

4.3 Overall Computation

In order to see how the different parameters influence the time for retrieving the overall rating for an ontology, we have based the computation on only the top review, the top 3 reviews and all reviews (just for worst-case considerations). For each of these settings, we base the computation on both local trust and global trust. Furthermore, we run each computation 300 times, to get more accurate average results. We measure the execution time for each of the 300 runs, and present the maximum time as well as average and minimum time needed for providing the result. It is intended to make use of the databases caching techniques, since they can also be exploited in real systems (how will be explained later). For the local trust based computation, we changed the user for which the results were computed in between the 3 different settings (top-1, top-3, all) so that results would have to be re-cached). The results can be found in figures 6, 7, 8, 11, 12 and 13.

The overall computation is one of the most important features of the TS-ORS and it is performed constantly at runtime. So here a quick response time is far more important than for the larger computations which are performed offline and only at dedicated time-points.

The results indicate that while the maximal time in case of a cache miss or for other system-specific reasons can be in the order of seconds (bare in mind that this is for all 12 ontologies), the more significant average is relatively independent of the amount of users or the number of reviews in the system. The only difference is that retrieving the result based on global trust is roughly 5-10 times faster than retrieving the result based on local trust. Is is due to the fact that for local trust more database queries have been made, if the top reviews cannot be found within the first query. The way the top reviews are found differs between local and global trust.

For global trust, retrieving the top reviews is simply one query to the global trust database with a weighing parameter alpha. Then an ordered result is retrieved that can be used for computation. In case of local trust, first the top locally trusted reviews are retrieved, then all possible remaining global trust results, and lastly the locally distrusted reviews.

Now considering a realistic scenario (e.g. the 10%10%20% Top 3 reviews), based on global trust the score for one ontology can be retrieved in roughly 3 ms and for local trust in roughly 8 ms. These values are more and less independent of the user size. Given that all 3 reviews are found in the first part of the local trust query, the results for local trust can be even less than 8 ms.

4.4 Review Retrieval

When a user wants to browse reviews for ontology–property combinations, it is important to retrieve them in a personalized order. So when a user is logged in, the reviews are retrieved in an order based on local trust of this user, otherwise they are ordered according to global trust. We have benchmarked retrieving all of the reviews in aforementioned personalized order for all ontology-property combinations. In figures 9 and 14, you can see the results both based on local trust and global trust. Again, we have run each task 300 times to compute accurate results.

This task will also often be requested at runtime, when a user is browsing different ontologies. Since all the reviews have to be ordered, results are better when there are not too many reviews. For a realistic scenario (e.g. the 10%10%20%), the time to retrieve the ordered reviews is about 0.7 and 1.5 ms for each ontology–properties based on global trust (depending on number of users and thus reviews). For retrieving the results based on local trust, between 2.7 ms and 7 ms are needed on the fast machine.

5 Results Learnt from Benchmark

Since many operations rely on a fast database, having good harddisks and enough caching enabled for the database is key to obtaining a good performance. In order to minimize query time at runtime, after each computation of trust, all ontology–property combinations could be queried once, to have the results stored in cache. This would lead to fast runtime response times.

It also seems to be sensible to use at least a dual-core machine with 2 threads for the trust computation, since the results are much fast than those obtained for a one-core solution. Depending on how important it is to take the latest user data into account, the frequency of overall recomputation can be increased or lowered. Amazon.com, for example, take 24 hours to take a trusts statement into account. In case a really fast recomputation is needed, the computation can be distributed among different machines, each containing a database filled with only the necessary information. Since the computation of trust is independent for all ontology–property combinations, in the most extreme case, you could use one machine per combination and later merge the results.

The runtime performance looks promising, since most of the time values will only be asked for preselected ontologies that were in the resultset of a user-query. In case this resultset is limited to e.g. 50 ontologies, the overall ratings could be generated in less than 500 ms. Also retrieving the reviews is done very quickly. So the overall performance is satisfying. Nevertheless, we will see how to optimize the code even further.

References

1. Sabou, M., Angeletou, S., d'Aquin, M., Barrasa, J., Dellschaft, K., Gangemi, A., Lehmann, J., Lewen, H., Maynard, D., Mladenic, D., Nissim, M., Peters, W., Pre-

sutti, V., Villazon, B.: D2.2.1 methods for selection and integration of reusable components from formal or informal user specifications. NeOn Project Deliverable D2.2.1, The Open University (MAY 2007)
2. d'Aquin, M., Lewen, H.: Cupboard – a place to expose your ontologies to applications and the community. In Aroyo, L., Traverso, P., Ciravegna, F., Cimiano, P., Heath, T., Hyvšnen, E., Mizoguchi, R., Oren, E., Sabou, M., Simperl, E.P.B., eds.: The Semantic Web: Research and Applications, 6th European Semantic Web Conference, ESWC 2009, Heraklion, Crete, Greece, May 31-June 4, 2009, Proceedings. Volume 5554 of Lecture Notes in Computer Science., Springer (MAY 2009) 913–918
3. Page, L., Brin, S., Motwani, R., Winograd, T.: The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford University, CA, USA (1998)